UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A208 767

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) **Ada Compiler Validation Summary Report:** DDC-I International A/S, DDC-I Ada Compiler System, Version 4.3, VAX 8530 Host and Target 881212W1.10011	5. TYPE OF REPORT & PERIOD COVERED 7 Dec. 1988 to 7 Dec. 1988
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Wright-Patterson AFB Dayton, OH, USA	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patterson AFB Dayton, OH, USA	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE
	13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Wright-Patterson AFB Dayton, OH, USA	15. SECURITY CLASS (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 if different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DDC-I International A/S, DDC-I Ada Compiler System, Version 4.3, Wright-Patterson AFB, VAX 8530 under VMS, Version 4.5 (Host) to VAX 8530 under VMS, Version 4.5 (Target), ACVC 1.10.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

UNCLASSIFIED

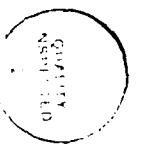
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AVF Control Number: AVF-VSR-212-0389
88-09-01-DDC

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 881212W1.10011
DDC International A/S
DDC-I Ada Compiler System, Version 4.3
VAX 8530 Host and Target

Completion of On-Site Testing:
7 December 1988

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503



Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: DDC-I Ada Compiler System, Version 4.3

Certificate Number: 881212W1.10011

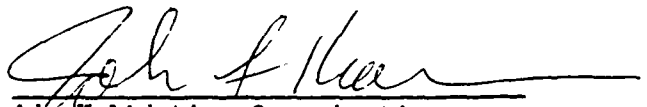
Host: VAX 8530 under VMS, Version 4.5

Target: VAX 8530 under VMS, Version 4.5

Testing Completed 7 December 1988 Using ACVC 1.10

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503



Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
William S. Richie
Acting Director
Department of Defense
Washington DC 20301

Ada Compiler Validation Summary Report:

Compiler Name: DDC-I Ada Compiler System, Version 4.3

Certificate Number: 881212W1.10011

Host: VAX 8530 under VMS, Version 4.5

Target: VAX 8530 under VMS, Version 4.5

Testing Completed 7 December 1988 Using ACVC 1.10

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
William S. Richie
Acting Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 7 December 1988 at Copenhagen, Denmark.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer which executes the code generated by the compiler.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be

customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: DDC-I Ada Compiler System, Version 4.3

ACVC Version: 1.10

Certificate Number: 881212W1.10011

Host Computer:

Machine:	VAX 8530
Operating System:	VMS, Version 4.5
Memory Size:	20 Megabytes

Target Computer:

Machine:	VAX 8530
Operating System:	VMS, Version 4.5
Memory Size:	20 Megabytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

(1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

(2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

(4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined types SHORT_INTEGER, LONG_INTEGER, and LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

(1) Some of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

(2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

(3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

(4) NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

(1) The method used for rounding to integer is round away from zero. (See tests C46012A..Z (26 tests).)

(2) The method used for rounding to longest integer is round away from zero. (See tests C46012A..Z (26 tests).)

(3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR. (See test C36003A.)

(2) NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

(3) NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components (in this test, the number of elements is not known at compile time). (See test C36202B.)

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array objects are declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

(6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

(7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

(8) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

f. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

(1) In the evaluation of a multi-dimensional aggregate, the test results indicate that all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

(2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

(1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

(1) Generic specifications and bodies can be compiled in separate compilations only if the specification and body are compiled before any instantiation of the generic. (See tests CA1012A, CA2009C, CA2009F, CA3011A, BC3204C, and BC3205D.)

j. Input and output.

(1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. However, USE_ERROR will be raised if an attempt is made to create a file with these types. (See tests AE2101C, EE2201D, and EE2201E.)

(2) The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

(3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

(4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, and CE2102V.)

(5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)

(6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

(7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

(8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

(9) Overwriting to a sequential file truncates to the last element written. (See test CE2208B.)

(10) Temporary sequential files are not given names. (See test CE2108A.)

(11) Temporary direct files are not given names. (See test CE2108C.)

(12) Temporary text files are not given names. (See test CE3112A.)

(13) More than one internal file can be associated with each external file for sequential files when reading only. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

(14) More than one internal file can be associated with each external file for direct files when reading only. (See tests CE2107F..H (3 tests), CE2110D and CE2111H.)

(15) More than one internal file can be associated with each external file for text files when reading only. (See tests CE3111A, CE3111B, CE3111D, CE3111E, CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 25 tests had been withdrawn because of test errors. The AVF determined that 552 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 23 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	122	1133	1792	17	30	46	3140
Inapplicable	6	6	536	0	4	0	552
Withdrawn	1	1	23	0	0	0	25
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	196	571	554	248	172	99	160	332	135	36	251	106	280	3140
N/A	17	78	126	0	0	0	6	1	2	0	2	279	41	552
Wdrn	0	1	0	0	0	0	0	1	0	0	0	19	4	25
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 25 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

A39005G	B97102E	CD2A62D	CD2A63B	CD2A63D	CD2A66B
CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D	CD2A76A
CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G	CD2B15C
CD7105A	CD7205C	CD7205D	CE2107I	CE3111C	CE3301A
CE3411B					

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 552 tests were inapplicable for the reasons indicated:

a. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests) C35705L..Y (14 tests)
C35706L..Y (14 tests) C35707L..Y (14 tests)

C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

b. C24113I, C24113J, and C24113K are not applicable because MAX_IN_LEN is exceeded.

c. C35508I, C35508J, C35508M, and C35508N are not applicable because they include enumeration representation clauses for BOOLEAN types in which the representation values are other than (FALSE => 0, TRUE => 1). Under the terms of AI-00325, this implementation is not required to support such representation clauses.

d. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

e. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 47.

f. C4A013B is not applicable because the evaluation of an expression involving 'MACHINE_RADIX applied to the most precise floating-point type would raise an exception; since the expression must be static, it is rejected at compile time.

g. B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

h. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

i. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

j. C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.

k. CA2009C, CA2009F, BC3204C and BC3205D are not applicable because this implementation does not permit compilation of generic bodies in in separate files after the instantiation of the generic.

l. The following 78 tests are not applicable because this implementation does not support address clauses.

CD5003B..I (8 tests)		CD5007B	CD5011A	CD5011B
CD5011C	CD5011D	CD5001E	CD5011F	CD5011G
CD5011H	CD5011I	CD5011K	CD5011L	CD5011M
CD5011N	CD5011O	CD5011Q	CD5011R	CD5011S
CD5012A	CD5012B	CD5012C	CD5012D	CD5012E

CD5012F	CD5012G	CD5012H	CD5012I	CD5012J
CD5012L	CD5012M	CD5013A	CD5013B	CD5013C
CD5013D	CD5013E	CD5013F	CD5013G	CD5013H
CD5013I	CD5013K	CD5013L	CD5013M	CD5013N
CD50130	CD5013R	CD5013S	CD5014A	CD5014B
CD5014C	CD5014D	CD5014E	CD5014F	CD5014G
CD5014H	CD5014I	CD5014J	CD5014K	CD5014L
CD5014M	CD5014N	CD50140	CD5014R	CD5014S
CD5014T	CD5014U	CD5014V	CD5014W	CD5014X
CD5014Y	CD5014Z			

m. This implementation does not support representation clauses for a derived type. Therefore the following 92 tests are not applicable:

AD1C04D	AD3015F	AD3015H	AD3015K	CD1C04A..C
CD1C04E	CD2A21C..D	CD2A22C..D	CD2A22G..H	CD2A23C..D
CD2A24C..D	CD2A24G..H	CD2A31C..D	CD2A32C..D	CD2A32G..H
CD2A41C..D	CD2A42C..D	CD2A42G..H	CD2A51C..D	CD2A52C..D
CD2A52G..H	CD2A53C..D	CD2A54C..D	CD2A54G..H	CD2A61A..B
CD2A61E..F	CD2A61I..J	CD2A62A..B	CD2A63A..D	CD2A64A..C
CD2A65A..C	CD2A66A	CD2A66C	CD2A71A..B	CD2A72A..B
CD2A74A..C	CD2A75A..C	CD2A81C..D	CD2A87A	CD2A91C..D
CD2D11B	CD3015A	CD3015B	CD3015D	CD3015E
CD3015G	CD3015I	CD3015J	CD3015L	CD4051A..D

n. This implementation does not support 'SIZE clause. Therefore the following 104 tests are not applicable:

A39005B	C87B62A	CD1009A..I	CD10090..Q	CD1C03A
CD2A21A..B	CD2A21E	CD2A22A..B	CD2A22E..F	CD2A22I..J
CD2A23A..B	CD2A23E	CD2A24A..B	CD2A24E..F	CD2A24I..J
CD2A31A..B	CD2A32A..B	CD2A32E..F	CD2A32I..J	CD2A41A..B
CD2A41E	CD2A42A..B	CD2A42E..F	CD2A42I..J	CD2A51A..B
CD2A51E	CD2A52A..B	CD2A52I..J	CD2A53A..B	CD2A53E
CD2A54A..B	CD2A54I..J	CD2A61C..D	CD2A61G..H	CD2A61K..L
CD2A62C	CD2A64D	CD2A65D	CD2A71C..D	CD2A72C..D
CD2A74D	CD2A75D	CD2A81A..B	CD2A81E..F	CD2A83A..C
CD2A83E	CD2A83F	CD2A84B..I	CD2A84K..N	CD2A91A..B
CD2A91E	ED2A26A	ED2A86A		

o. This implementation does not support 'SMALL clause. Therefore the following tests are not applicable:

A39005E	C87B62C	CD1009L	CD1C03F	CD2D11A
CD2D13A	ED2A56A			

p. CD4041A is not applicable because this implementation does not support alignment clauses for record representation clauses.

q. This implementation supports OPEN, CREATE, and RESET with IN_FILE and OUT_FILE modes for SEQUENTIAL_IO. Therefore the following 6 tests are not applicable:

    CE2102E CE2102N..Q CE2105A

r. This implementation supports CREATE, OPEN, and RESET with IN_FILE, OUT_FILE, and INOUT_FILE modes for DIRECT_IO. Therefore the following 9 tests are not applicable:

    CE2102F CE2102J CE2102R..W CE2105B

s. CE2107B, CE2107E, and CE2110B are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

t. CE2107C..D (2 tests), CE2107H, CE2107L, CE2108B, and CE2108D are not applicable because temporary files are not given names in this implementation.

u. CE2107G, CE2110D, and CE2111H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

v. CE2111D is not applicable because this implementation does not allow two internal files associated with the same external file opened for writing.

w. CE3102E is not applicable because CREATE with mode IN_FILE is allowed.

x. CE3102F is inapplicable because text file RESET is supported by this implementation.

y. CE3102G is inapplicable because text file deletion of an external file is supported by this implementation.

z. CE3102I is inapplicable because text file CREATE with OUT_FILE mode is supported by this implementation.

aa. CE3102J is inapplicable because text file OPEN with IN_FILE mode is supported by this implementation.

ab. CE3102K is inapplicable because text file OPEN with OUT_FILE mode is not supported by this implementation.

ac. CE3111B, CE3111D, CE3111E, CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

ad. In test CE3804G a float literal which is not a model number is written to a text file and later read again. Since the literal, -3.525, is not a model number, the value read may not equal the float literal.

ae. EE2401D is not applicable because USE_ERROR is raised when trying to create a file with unconstrained array types.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 24 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B33301B	B37302A	B55A01A	B67001H	BA1101B
BC1109A	BC1109C	BC1109D	BC3009A	

The following modifications were made to compensate for legitimate implementation behavior:

a. At the recommendation of the AVO, LONG_INTEGER was substituted for INTEGER in test AD7006A and LONG_INTEGER'IMAGE was substituted for INTEGER'IMAGE in test ED7006C since SYSTEM.MEMORY_SIZE is outside the range of INTEGER for this implementation.

b. In this implementation, the value of PARENT'STORAGE_SIZE, in C34007A, C34007D, C34007G, C34007J, C34007M, C34007P, and C34007S, and the value of JUST_LIKE_LINK'STORAGE_SIZE, in C87B62B, is undefined if no length clause has specified its value. Therefore, such a phrase was added to these tests.

c. At the recommendation of the AVO, a "PRAGMA ELABORATE (REPORTS)" was added at appropriate points in tests ED7004B, ED7005C, ED7005D, ED7006C, ED7006D to ensure that the elaboration of the routines in package REPORT takes place before these routines are

called.

Two tests, C34005G and CE3804G, execute and produce "FAILED" messages; the AVO has recommended that these tests be evaluated to determine conforming behavior that was not anticipated by the test. As recommended by the AVO, C34005G is graded as passed if the only reported failures relate to 'SIZE, which is the case for the DDC compiler. As recommended by the AVO, CE3804G is graded as not-applicable to the implementation if the float literal, which is not a model number, does not equal the value after the literal is written to a file and read back again.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the DDC-I Ada Compiler System, Version 4.3 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the DDC-I Ada Compiler System, Version 4.3 compiler using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	VAX 8530
Host operating system:	VMS, Version 4.5
Target computer:	VAX 8530
Target operating system:	VMS, Version 4.5
Compiler:	DDC-I Ada Compiler System, Version 4.3

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer.

TEST INFORMATION

After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the VAX 8530. Results were printed from the VAX 8530.

The compiler was tested using command scripts provided by DDC International A/S and reviewed by the validation team. The compiler was tested using all default option settings except for the following:

OPTION	EFFECT
/OPTIMIZE	Optimizes the generated code.
/LIST	Generates a compilation list file.

The "ADA" command invokes the DDC-I Ada Compiler System. Qualifiers serve to differentiate the different functions provided.

The Ada system is documented in:

DDC-I Ada Compiler System
Version 4.3 for VAX/VMS
User's Guide
1 January 1988

Format:

ADA file-spec

Parameters

file-spec

Specifies an Ada source file containing compilation units to be compiled. If a file type for an input file is not specified, the default file type ADA is used.

No wildcard characters are allowed in the file specification.

Qualifiers

/LIST,
/NOLIST (DEFAULT)

Controls whether the compiler creates a list file.

The name of the listing file will be that of the invocation parameter with the file type LIS.

If /NOLIST is active, no source listing is produced regardless of any LIST pragmas in the compilation or any diagnostic message produced.

/XREF,
/NOXREF (DEFAULT)

Controls whether the compiler generates a cross-reference listing.

If /XREF is specified and no severe or fatal errors are found during the compilation, the cross-reference listing is appended to the listing file (see /LIST).

/LIBRARY = file-id,
/LIBRARY = ADA_LIBRARY (DEFAULT)

This qualifier specifies the current sublibrary and thereby also the current program library which consists of sublibrary and its ancestor sublibraries.

If the qualifier is omitted, the sublibrary designated by the logical name ADA_LIBRARY is used as the current sublibrary.

/CONFIGURATION_FILE=file-spec

This qualifier specifies the configuration file to be used by the compiler in the current compilation.

If the qualifier is omitted, the configuration file designated by the logical name ADA_CONFIG is used by default.

The configuration file determines some characteristics of the compiler as the maximum input line length, the format of error messages, the format of the list file and specifies the maximum number of errors the compiler will report before a compilation is aborted.

/OPTIMIZE,
/OPTIMIZE=(<list of optimization>),
/NOOPTIMIZE (DEFAULT)

This qualifier specifies whether any optimizations should be performed on the generated code.

/OPTIMIZE	Causes all optimizations to be performed.
/OPTIMIZE=CSE	Normal common subexpression elimination is performed.
/OPTIMIZE=STACK_HEIGHT	The use of temporary variables in expression evaluations is minimized.
/OPTIMIZE=BLOCK	The generation of scope information by the code generator is minimized.
/OPTIMIZE=FCT2PROC	Certain functions are transformed into procedures.

/OPTIMIZE=PEEP	Extensive peep hole optimization will be performed.
/OPTIMIZE=REORDERING	The compiler will try to reorder an aggregate with named component association into an aggregate with positional component association. It will also reorder named parameter association into an aggregate with positional parameter association.

/CHECK (DEFAULT),
/NOCHECK

This qualifier specifies whether code should be generated to do the checking described in ARM. /NOCHECK specifies that no checking at all should be performed at run time.

/TRACEBACK (DEFAULT), /NOTRACEBACK

This qualifier specifies whether the code generator should generate trace-back information. The trace-back information enables a program to give a trace if an unhandled exception occurs. The trace-back information can be removed if the user wants to save data space.

/SAVE_SOURCE (DEFAULT), /NOSAVE_SOURCE

This qualifier specifies whether the source text is stored in the program library. In case that the source text contains several compilation units, the source text for each compilation unit is stored in the program library.

The source texts stored in the program library can be extracted using the PLU command TYPE.

/PROGRESS, /NOPROGRESS (DEFAULT)

This qualifier causes the compiler to output data about which pass the compiler is currently running. Default is not to output this information.

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Copenhagen, Denmark and was completed on 7 December 1988.

APPENDIX A

DECLARATION OF CONFORMANCE

DDC International A/S has submitted the following Declaration of Conformance concerning the DDC-I Ada Compiler System, Version 4.3 compiler.

2 Declaration of Conformance

Compiler Implementor: DDC International A/S
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503 Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name: DDC-I Ada Compiler System	Version: 4.3
Host Architecture ISA: VAX 8530	OS&VER #: VMS 4.5
Target Architecture ISA: VAX 8530	OS&VER #: VMS 4.5

Implementors Declaration

I, the undersigned, representing DDC International A/S, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler listed in this declaration. I declare that DDC International A/S is the owner of record of the Ada language compiler listed above and, as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for the Ada language compiler listed in this declaration shall be made only in the owner's corporate name.

DDC International A/S Date: 12 October 1994

Jørgen Bundgaard

Owner's Declaration

I, the undersigned, representing DDC International A/S, take full responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that the Ada compiler listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

DDC International A/S Date: 12 October 1994

Jørgen Bundgaard

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the DDC-I Ada Compiler System, Version 4.3 compiler, as described in this appendix, are provided by DDC International A/S. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of appendix F, are:

```
package STANDARD is

   ...

     type INTEGER is range -32_768 .. 32_767;
     type SHORT_INTEGER is range -128 .. 127;
     type LONG_INTEGER is range -2_147_483_648 .. 2_147_483_647;

     type FLOAT is digits 6 range -1.70141E+38 .. 1.70141E+38;
     type LONG_FLOAT is digits 15 range
           -16#7.FFFF_FFFF_FFFF#E255 .. 16#7.FFFF_FFFF_FFFF#E255;

     type DURATION is delta 2**(-14) range -131_072.00000 .. 131_071.00000;

   ...

end STANDARD;
```

F Appendix F of the Ada Reference Manual

F.0 Introduction

This appendix describes the implementation-dependent characteristics of the DDC-I VAX/VMS Ada Compiler, as required in the Appendix F frame of the Ada Reference Manual (ANSI/MIL-STD-1815A).

F.1 Implementation-Dependent Pragmas

Currently there is one implementation defined pragma:

Pragma INTERFACE SPELLING

The pragma has the form:

```
pragma INTERFACE_SPELLING (routine_name, string);
```

It is used to establish the connection between a subprogram name mentioned in a pragma INTERFACE (routine_name) and an external name (given by the string). The external name is the name the linker will look for when linking the program. This is useful when the external name does not conform with the lexical rules for identifiers in Ada (e.g the VAX/VMS library routines called LIBS*).

F.2 Implementation-Dependent Attributes

No implementation-dependent attributes are defined for the VAX/VMS version.

F.3 Package SYSTEM

The specification of the package SYSTEM:

```
package SYSTEM is

   type ADDRESS          is access INTEGER;
   subtype PRIORITY      is INTEGER range 0..15;
   type NAME             is (VAX11);
   SYSTEM_NAME:          constant NAME  := VAX11;
   STORAGE_UNIT:         constant       := 8;
   MEMORY_SIZE:          constant       := 2048 * 1024;
   MIN_INT:              constant       := -2_147_483_647-1;
   MAX_INT:              constant       := 2_147_483_647;
   MAX_DIGITS:           constant       := 15;
   MAX_MANTISSA:         constant       := 31;

   FINE_DELTA:           constant       := 2#1.0#E-31;
   TICK:                 constant       := 0.000_001;

   type interface_language is (VMS);

end SYSTEM;
```

F.4 Representation Clauses

In general, no representation clauses may be given for a derived type. The representation clauses that are accepted for non-derived types are described in the following:

Length Clause

The compiler accepts two kinds of length clauses, specifying either the number of storage units to be reserved for a collection, or the number of storage units to be reserved for an activation of a task.

Enumeration Representation Clause

Enumeration representation clauses may specify representations only in the range of the predefined type INTEGER.

Record Representation Clause

Alignment clauses in record representation clauses are not supported.

Component clauses in record representation clauses are supported when the following restrictions are fulfilled:

- components of a discrete type other than LONG_INTEGER are packed in at most 31 bits.

- the bit width given must match the subtype of the component, i.e. all values in the subtype should be representable by using the given number of bits.

- arrays with a discrete element type other than LONG_INTEGER are packed. Elements are of size 1,2,4,8 or 16 bit.

- components of non discrete type (not array types with discrete element type) or LONG_INTEGER must start at a storage unit boundary and must be specified with correct size.

If the record type contains components not covered by a component clause, they are allocated consecutively after the component with the highest offset from the start of the record. Allocation of a record component without a component clause is always aligned on a storage unit boundary. Holes created because of component clauses are not otherwise utilized by the compiler.

F.5 Implementation-Dependent Names for Implementation-Dependent Components

None defined by the compiler.

F.6 Address Clauses

Not supported by the compiler.

F.7 Unchecked Conversion

Unchecked conversion is only allowed between objects of the same "size". For dynamic arrays this will be checked at run-time. Unchecked conversion between types where at least one is an unconstrained array type is not allowed. This is the only restriction imposed on unchecked conversion.

F.8 Input-Output Packages

The implementation supports all requirements of the Ada language. It is an effective interface to the VAX/VMS file system, and in case of text input-output also an effective interface to the VAX/VMS terminal driver.

This section describes the functional aspects of the interface to the VAX/VMS file system and terminal driver. Certain portions of this section are of special interest to the system programmer who needs to control VAX/VMS specific Input-Output characteristics via Ada programs.

The section is organized as follows.

Subsection numbers refer to the equivalent subsections in Chapter 14 of the ARM. Only subsections of interest to this section are included.

The Ada Input-Output concept as defined in Chapter 14 of the ARM does not constitute a complete functional specifications of the Input-Output packages. Some aspects are not discussed at all, while others are deliberately left open to an implementation.

These gaps are filled in the appropriate subsections and summarized in subsection F.8.a.

The reader should be familiar with

[DoD 83] - The Ada language definition

and certain sections require that the reader is familiar with

[DEC 84a] - Guide to VAX/VMS File Applications

[DEC 84b] - Record Management Services

[DEC 85] - VAX/VMS I/O Users Reference Manual

F.8.1 External Files and File Objects

An external file is either any VAX/VMS file residing on a file-structured device (disk, tape), a record structured device (terminal, lineprinter), or a virtual software device (mailbox). ARM 14.1(1).

Identification of an external file by a string (the NAME parameter) is described in subsection F.8.2.1.

System-dependent characteristics (the FORM parameter) is described in subsection F.8.2.1

An external file created on a file-structured device will exist after program termination, and may be accessed later from an Ada program, except if the file is a temporary file created by using an empty name parameter. If files corresponding to the external file have not been closed, the external file will also exist upon program completion, and the contents will be the same as if the files had been closed prior to program completion. See further F.8.3. ARM 14.1(7).

Input-Output of access types will cause input-output of the access value [Dod 83] 14.1(7).

Sharing of an external file is, when using the default system-dependent characteristics, handled as described in the following.

When a file is associated with an external file using the Record Management Services (RMS), and the file is opened with mode IN_FILE, the implementation will allow the current process and other processes to open files associated with the same external file (e.g. as IN_FILE in an Ada program).

When a file is opened with mode INOUT_FILE or OUT_FILE no file sharing is allowed when using RMS. In particular, trying to gain write access to an external file shared by other files, by OPEN or RESET to mode INOUT_FILE or OUT_FILE will raise USE_ERROR.

When a text file is associated with a terminal device, using the Queue I/O System Services (QIO), there are no restrictions on file sharing.

F.8.2 Sequential and Direct Files

When dealing with sequential and direct input-output only RMS files are used.

In this section, a description of the basic file-mapping is given.

Basic file-mapping concerns the relation between Ada files and (formats of) external RMS files, and the strategy for accessing the external files. When creating new files (with the CREATE procedure), there is a unique mapping onto a RMS file format, the preferred file format. When opening an existing external file (with the OPEN procedure), the mapping is not unique; i.e. several external file formats other than preferred for CREATE may be acceptable. In subsection F.8.2.1 the preferred and acceptable formats are described for sequential and direct input-output. In subsection F.8.3.1 the preferred and acceptable formats are described for text input-output.

F.8.2.1 File Management

This subsection contains information regarding file management:

- Description of preferred and acceptable formats for sequential and direct input-output.

- The NAME parameter.

- The FORM parameter.

- File access.

Preferred and Acceptable Formats

The preferred and acceptable formats for sequential and direct input-output, are described using RMS notation and abbreviations [DEC 84b]. ES is used to denote the element size, i.e. the number of bytes occupied by the element type, or, in case of a varying size type, the maximum size (which must be determinable at the point of instantiation from the value of the SIZE attribute for the element type).

It should be noted that the latter means a type definition like:

```
type large_type is array( integer <> ) of integer;
```

would be mapped onto an element size greater than the maximum allowed size (32 k byte).

SEQUENTIAL_IO:

An element is mapped into a single record of the external file, or if block-io is used, a number of consecutive virtual blocks of 512 bytes. ES must not be greater than 32767, otherwise USE_ERROR is raised.

CREATE - preferred file format

- ORG=SEQ, RFM=FIX, MRS=ES
  (note: read and write operations will be done by BLOCK IO if element size is a multiple of 512 bytes)

OPEN - acceptable formats

- ORG=REL, RFM=FIX, MRS=ES

- ORG=SEQ, RFM=FIX, MRS=ES

- ORG=SEQ, RFM=VAR

- ORG=SEQ, RFM=UDF
  (note: BLOCK IO will be used)

  (note: a RESET operation to OUT_FILE mode will give a USE_ERROR exception, as it is not possible to empty a file of this format).

The detailed setting of the control blocks for sequential_IO is given below. Note that the user-provided form parameter will override the default specified settings, when used with OPEN or CREATE.

Also note that, when an Ada program contains tasks, asynchronous I/O will be used (ROP = <ASY>).

The following shows the initial setting for OPEN and CREATE (unspecified fields in the control blocks will be cleared to zero).

```
FAB:
  ALQ = 12
  DEQ = 6
  DNM = <.DAT>
  FAC = for block-io, IN_FILE:     <BRO,GET>
        for block-io, OUT_FILE:    <BRO,PUT,UPD,DEL,TRN>
        otherwise,    IN_FILE:     <GET>
        otherwise,    OUT_FILE:    <PUT,UPD,DEL,TRN>
  FNM = name parameter
  FOP = non-empty name parameter:          <MXV,SQO>
        empty name parameter to CREATE: <MXV,SQO,TMP>
  MRS = element size (in bytes)
  NAM = address of name-block
  ORG = SEQ
  RAT = <CR>
  RFM = FIX
  SHR = for IN_FILE:   <GET>
        for OUT_FILE:  <NIL>
  XAB = address of XABFHC block

RAB:
  FAB = address of FAB block
  KBF = address of internal longword
  KSZ = 4
  RAC = SEQ
  ROP = for block-io:  <BIO>
        otherwise:     <UIF>
NAM:
  RSA = address of internal 255 byte buffer
  RSS = 255

XABFHC:
  NXT = 0
```

DIRECT_IO:

An element is mapped into a single record of the external file, or if block io is used, the smallest possible number of consecutive virtual blocks of 512 bytes. ES must not be greater than 32767, otherwise USE_ERROR will be raised.

CREATE - preferred file format

- if element size is not a multiple of 512: ORG=REL, RFM=FIX, MRS=ES

- if element size is a multiple of 512: ORG=SEQ, RFM=FIX, MRS=ES
  (note: read and write operations will be done by BLOCK IO)

OPEN - acceptable formats

- ORG=REL, RFM=FIX, MRS=ES

- ORG=SEQ, RFM=FIX, MRS=ES
  (note: if element size is a multiple of 512, BLOCK IO will be used)

- ORG=SEQ, RFM=UDF
  (note: BLOCK IO will be used)

The detailed setting of the control blocks for direct_IO is given below. Note that the user-provided form parameter will override the default specified settings, when used with OPEN or CREATE.

Also note that, when an Ada program contains tasks, asynchronous I/O will be used (ROP = <ASY>).

The initial setting for OPEN and CREATE (unspecified fields in the control blocks will be cleared to zero) follows:

```
FAB:
  ALQ = 12
  DEQ = 6
  DNM = <.DAT>
  FAC = for IN_FILE:  <GET>
        for OUT_FILE: <GET,PUT,UPD,DEL,TRN>
  FNM = name parameter

  FOP = non-empty name parameter:        <MXV,SQO>
        empty name parameter to CREATE: <MXV,SQO,TMP>
  MRS = 512
  NAM = address of name-block
  ORG = SEQ
  RAT = <CR>
  RFM = VAR
  SHR = for IN_FILE:  <GET>
        for OUT_FILE: <NIL>
  XAB = address of XABFHC block
```

```
RAB:
  FAB = address of FAB block
  KBF = address of internal longword
  KSZ = 4
  RAC = SEQ
  ROP = <>
  UBF = address of internal 512 byte buffer
  USZ = 512

NAM:
  RSA = address of internal 255 byte buffer
  RSS = 255

XABFHC:
  NXT = 0
```

Name Parameter

The name parameter, when non null, must be a valid VAX/VMS file specification referring to a file-structured device; a file with that name will then be created.

For a null name parameter, the process' current directory and device must designate a directory on a disk device; a temporary, unnamed file marked for deletion will then be created in that directory. The file will be deleted after closing it, or, if not closed when the program terminates. ARM 14.2.1(3).

Form Parameter

The FORM string parameter that can be supplied to any OPEN or CREATE procedure is for controlling the external file properties, such as physical organization, allocation etc. In the present implementation this has been achieved by accepting form parameters that specify setting of fields in the RMS control blocks FAB and RAB, used for all open files. This scheme is rather general in that it accepts all settings of the FAB and RAB fields. It opens for modifications of the behaviour required by the Arm, such as being able to open a file for appending data to it. Furthermore, a form parameter for accessing mailboxes is provided.

The following fields can currently not be set explicitly:

```
FAB:
  FNA, FNS (are set by the NAME parameter of OPEN or
            CREATE)
  DNA, DNS (can be set by DNM=/.../)
```

The syntax of the form parameter is as follows:

```
form_parameter ::= [ param { , param } ]

param          ::= number_param
                 | string_param
                 | quotation_param
                 | mask_param

number_param   ::= keyword = number
number         ::= digit { digit }
digit          ::= 0 | 1 | ... | 9
string_param   ::= keyword = string
string         ::= / {any character other than slash) /

quotation_param::= keyword = specifier

mask_param     ::= clear_bits
                 | set_bits
                 | define_whole_field

clear_bits     ::= keyword - mask
set_bits       ::= keyword + mask
define_whole_field
               ::= keyword = mask
mask           ::= < [ specifier { , specifier } ] >

keyword        ::= letter letter letter
specifier      ::= letter letter letter [ letter letter ]

letter         ::= A | B | ... | Z | a | b | ... | z
```

Notes:

- all space characters are ignored.
- string parameters are converted to uppercase.
- all keywords and specifiers are 3- or 5-letter words, like the RMS assembly level interface symbolic names. The only exceptions are the RAT=<CR> specifier, which in this implementation must be specified as CAR rather than CR, and the RAB CTX field keyword, which must be specified as CON. There are only 2 5-letter words: the specifiers STMCR and STMLF.

The semantics of the form parameter is (except for the mailbox parameter) to modify the specified FAB and RAB fields just prior to actually calling RMS to open or create a file, i.e. the form parameter overrides the default conventions provided by this implementation (ARM section F.5.4). The form parameter is interpreted left to right, and it is legal to respecify fields; in particular a mask field may be manipulated in several turns.

Note that there is no way of modifying fields after an RMS open or create service, in particular it is not possible to set RAB fields on a per record operation basis.

The modifications made are those to be expected from the textually corresponding RMS macro specifications. However, the clear_bits and set_bits are particular to this implementation: They serve to either clear individual mask specifiers set by the implementation default, or to set mask specifiers in addition to those specified by the implementation default, respectively.
The mailbox parameter can be either

```
MBX=TMP
```

or

```
MBX=PRM
```

It applies to CREATE only, and causes either a temporary or a permanent mailbox to be created. The NAME parameter will be used to establish a logical name for the mailbox, unless an empty string is specified (in this case, no logical name will be established).

Note that the implementation does in no way check that the form parameter supplied is at all reasonable. The attitude is "you asked for it, you got it". It is discouraged, if other procedures than OPEN, CREATE, and CLOSE will be called, to set ORG, RAC, MRS, NAM, FOP=<NAM>. It is generally discouraged to set XAB.

Examples:

```
-- create a text file
create(file, out_file, "DATA.TXT");

-- create a temporary text file which will be deleted
      after completion of the main program
create(file, out_file);

-- create an empty stream format text file
create(file,out_file,"DATA.DAT","ORG=SEQ,RFM=STMLF");

-- create a very big file:
create(file,out_file,"DATA.DAT","ALQ=2048,DEQ=256");

-- create a temporary mailbox:
create(file,out_file,"HELLO","MBX=TMP");

-- open a mailbox; at reading, do not wait for
      messages:
open(file,in_file,"HELLO","ROP+<TMO>,TMO=0");
```

File Access

The OPEN and CREATE procedures utilize the normal RMS defaulting mechanism to determine the exact file to open or create.

Device and directory (when not specified) defaults to the process' current device (SYS$DISK) and directory.

The version number (when not specified), defaults for OPEN to highest existing, or for CREATE, one higher than the highest existing, or 1 when no version exists.

The implementation provides .DAT as the default file type.

External files, which are not to be accessed via block-io (as described in formats), will be accessed via standard RMS access methods. For SEQUENTIAL_IO, sequential record access mode will be used. For DIRECT_IO, random access by record number will be used.

Creation of a file with mode IN_FILE will raise USE_ERROR, when referring to an RMS file.

For sequential and direct io, files created by SEQUENTIAL_IO for a given type T, may be opened (and processed) by DIRECT_IO for the same type and vice-versa. In the latter case, however, the function END_OF_FILE (14.2.2(8)) may fail to produce TRUE in cases where the file has been written at random, leaving "holes" in the file. See ARM 14.2.1(7).

For a sequential or text file associated with an RMS file, a RESET operation to OUT_FILE mode will cause deletion of any elements in the file, i.e. the file is emptied. Likewise, a sequential file or text file opened (by OPEN) with mode OUT_FILE, will be emptied. For any other RESET operation, the contents of the file is not affected.

For a text file, any RESET operation will cause USE_ERROR to be raised, when QIO services are used.

F.8.2.2 Sequential Input-Output

The implementation omits type checking for DATA_ERROR, in case the element type is of an unconstrained type, ARM 14.2.2(4), i.e.:

```
... f : FILE_TYPE
type et  is 1..100;
type eat is array( et range <> ) of integer;
```

```
X : eat( 1..2 );
Y : eat( 1..4 );
...
-- write X, Y:

write( f, X); write( f, Y); reset( f, IN_FILE);

-- read X into Y and Y into X:

read( f, Y); read( f, X);
```

This should have given DATA_ERROR, but will instead give undefined values in the last 2 elements of Y.

F.8.2.3 Specification of the Package Sequential IO

```
with BASIC_IO_TYPES;

with IO_EXCEPTIONS;

generic

   type ELEMENT_TYPE is private;

package SEQUENTIAL_IO is

  type FILE_TYPE is limited private;

  type FILE_MODE is (IN_FILE, OUT_FILE);

-- File management

   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE := OUT_FILE;
                    NAME : in     STRING    := "";
                    FORM : in     STRING    := "");

   procedure OPEN  (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE;
                    NAME : in     STRING;
                    FORM : in     STRING := "");

   procedure CLOSE (FILE : in out FILE_TYPE);

   procedure DELETE(FILE : in out FILE_TYPE);

   procedure RESET (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE);
```

```
   procedure RESET (FILE : in out FILE_TYPE);

   function MODE   (FILE : in FILE_TYPE) return FILE_MODE;

   function NAME   (FILE : in FILE_TYPE) return STRING;

   function FORM   (FILE : in FILE_TYPE) return STRING;

   function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

-- input and output operations

   procedure READ  (FILE : in     FILE_TYPE;
                    ITEM :    out ELEMENT_TYPE);

   procedure WRITE (FILE : in FILE_TYPE;
                    ITEM : in ELEMENT_TYPE);

   function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

-- exceptions

   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

private

   type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end SEQUENTIAL_IO;
```

F.8.2.4 Direct Input-Output

The implementation omits type checking for DATA_ERROR, in case the element type is of an unconstrained type, [Dod 83] 14.2.4(4), see F.8.2.2.

F.8.2.5 Specification of the Package Direct IO

```
with BASIC_IO_TYPES;
with IO_EXCEPTIONS;

generic

   type ELEMENT_TYPE is private;

package DIRECT_IO is


   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);

   type COUNT is range 0..LONG_INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;


-- File management

   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE  := INOUT_FILE;
                    NAME : in     STRING     := "";
                    FORM : in     STRING     := "");

   procedure OPEN  (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE;
                    NAME : in     STRING;
                    FORM : in     STRING    := "");

   procedure CLOSE (FILE : in out FILE_TYPE);

   procedure DELETE(FILE : in out FILE_TYPE);

   procedure RESET (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE);

   procedure RESET (FILE : in out FILE_TYPE);

   function MODE   (FILE : in FILE_TYPE) return FILE_MODE;

   function NAME   (FILE : in FILE_TYPE) return STRING;

   function FORM   (FILE : in FILE_TYPE) return STRING;

   function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

-- input and output operations
```

```
   procedure READ  (FILE : in     FILE_TYPE;
                    ITEM :    out ELEMENT_TYPE;
                    FROM : in     POSITIVE_COUNT);
   procedure READ  (FILE : in     FILE_TYPE;
                    ITEM :    out ELEMENT_TYPE);

   procedure WRITE (FILE : in FILE_TYPE;
                    ITEM : in ELEMENT_TYPE;
                    TO   : in POSITIVE_COUNT);
   procedure WRITE (FILE : in FILE_TYPE;
                    ITEM : in ELEMENT_TYPE);

   procedure SET_INDEX(FILE : in FILE_TYPE;
                       TO   : in POSITIVE_COUNT);

   function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;

   function SIZE (FILE : in FILE_TYPE) return COUNT;

   function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;


-- exceptions

   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

private

  type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end DIRECT_IO;
```

F.8.3 Text Input-Output

When utilizing text input-output, RMS is used when an external file is residing on a file-structured device, or is a virtual software device. When an external file that is a terminal device is opened or created, the queue I/O services (QIO) are used by default.

If a text file of mode OUT_FILE corresponds to an external RMS file, the external file will also exist upon program completion, and a pending linebuffer will be flushed before the text file is closed.

F.8.3.1 File Management

This subsection contains information regarding file management, where it differs from the file management described in F.8.2.1.

- Description of preferred and acceptable formats for text input-output.
- The FORM parameter.
- File access.

Preferred and Acceptable Formats

Lines of text are mapped into records of external files.

For output, the following rules apply.

The Ada line terminators and file terminators are never explicitly stored (however, for stream format files, RMS forces line terminators to trail each record). Page terminators, except the last, are mapped into a form feed character trailing the last line of the page. (In particular, an empty page (except the last) is mapped into a single record containing only a form feed character). The last page terminator in a file is never represented in the external file. It is not possible to write records containing more than 512 characters. That is, the maximum line length is 511 or 512, depending on whether a page terminator (form feed character) must be written or not. If output is more than 512 characters, USE_ERROR will be raised.

On input, a FF trailing a record indicates that the record contains the last line of a page and that at least one more page exists. The physical end of file indicates the end of the last page.

CREATE - preferred file format

- ORG=SEQ, RFM=VAR, MRS=512

OPEN - acceptable file formats

- all formats except
  - ORG=IDX
  - RFM=UDF

(Note: for stream files (RFM=STM...) any sequence of the LF, CR, and VT control characters at the end of a line will be stripped off at input. At output, line terminators will be provided by RMS defaults). (Note: input of any record containing more than 512 characters will raise a USE_ERROR exception).

The detailed setting of the control blocks for TEXT_IO is given below. Note that the user-provided form parameter will override the default specified settings, when used with OPEN or CREATE.

Also note that, when an Ada program contains tasks, asynchronous I/O will be used. When RMS files ROP = <ASY>, or asynchronous QIO when terminal devices.

The following shows the initial setting for OPEN and CREATE (unspecified fields in the control blocks will be cleared to zero):

```
FAB:
  ALQ = 12
  DEQ = 6
  DNM = <.DAT>
  FAC = for IN_FILE:      <GET>
        for OUT_FILE:     <GET,PUT,UPD,DEL,TRN>
  FNM = name parameter
  FOP = non-empty name parameter              <MXV,SQO>
        empty name parameter to CREATE:       <MXV,SQO,TMP>
  MRS = 512
  NAM = address of name-block
  ORG = SEQ
  RAT = <CR>
  RFM = VAR
  SHR = for IN_FILE:      <GET>
        for OUT_FILE:     <NIL>
  XAB = address of XABFHC block

RAB:
  FAB = address of FAB block
  KBF = address of internal longword
  KSZ = 4
  RAC = SEQ
  ROP = <>
  UBF = address of internal 512 byte buffer
  USZ = 512
```

```
NAM:
  RSA = address of internal 255 byte buffer
  USZ = 255

XABFHC:
  NXT = 0
```

Form parameter

If any form parameter, except for the empty string or a string containing only blanks, is supplied to OPEN or CREATE, RMS services will always be used. In this case, the file operations on external files as terminal-devices will use buffered input- output.

File access

External RMS files are accessed via sequential record access methods.

Files associated with terminal devices, using QIO services, do not contain page terminators. This means that calling SKIP_PAGE will raise USE_ERROR. Furthermore, trying to RESET a file in this category will cause USE_ERROR.

Files associated with the same external file, using QIO services, share the standard values (page-, line, and column-number), e.g. standard values for STANDARD_OUTPUT are implicitly updated after reading from STANDARD_INPUT.

F.8.3.10 Specification of the Package Text IO

```
with BASIC_IO_TYPES;
with IO_EXCEPTIONS;
package TEXT_IO is

   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, OUT_FILE);

   type    COUNT is range 0 .. LONG_INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
   UNBOUNDED: constant COUNT:= 0; -- line and page length

   subtype FIELD          is INTEGER range 0 .. 35;

   subtype NUMBER_BASE    is INTEGER range 2 .. 16;
```

```
type TYPE_SET is (LOWER_CASE, UPPER_CASE);

-- File Management

  procedure CREATE (FILE   : in out FILE_TYPE;
                    MODE   : in     FILE_MODE := OUT_FILE;
                    NAME   : in     STRING    := "";
                    FORM   : in     STRING    := ""
                   );

  procedure OPEN   (FILE   : in out FILE_TYPE;
                    MODE   : in     FILE_MODE;
                    NAME   : in     STRING;
                    FORM   : in     STRING    := ""
                   );

procedure CLOSE  (FILE   : in out FILE_TYPE);
procedure DELETE (FILE   : in out FILE_TYPE);
procedure RESET  (FILE   : in out FILE_TYPE;
                  MODE   : in     FILE_MODE);
procedure RESET  (FILE   : in out FILE_TYPE);

function  MODE   (FILE   : in FILE_TYPE) return FILE_MODE;
function  NAME   (FILE   : in FILE_TYPE) return STRING;
function  FORM   (FILE   : in FILE_TYPE) return STRING;

function  IS_OPEN(FILE   : in FILE_TYPE) return BOOLEAN;

-- Control of default input and output files

procedure  SET_INPUT   (FILE : in FILE_TYPE);
procedure  SET_OUTPUT  (FILE : in FILE_TYPE);

function   STANDARD_INPUT   return FILE_TYPE;
function   STANDARD_OUTPUT  return FILE_TYPE;

function   CURRENT_INPUT    return FILE_TYPE;
function   CURRENT_OUTPUT   return FILE_TYPE;


-- specification of line and page lengths

procedure SET_LINE_LENGTH  (FILE : in FILE_TYPE;
                            TO   : in COUNT);
procedure SET_LINE_LENGTH  (TO   : in COUNT);

procedure SET_PAGE_LENGTH  (FILE : in FILE_TYPE;
                            TO   : in COUNT);
procedure SET_PAGE_LENGTH  (TO   : in COUNT);
```

```
function  LINE_LENGTH      (FILE : in FILE_TYPE) return
                                                 COUNT;
function  LINE_LENGTH                            return
                                                 COUNT;

function  PAGE_LENGTH      (FILE : in FILE_TYPE) return
                                                 COUNT;
function  PAGE_LENGTH                            return
                                                 COUNT;

-- Column, Line, and Page Control

   procedure NEW_LINE    (FILE    : in FILE_TYPE;
                          SPACING : in POSITIVE_COUNT := 1);
   procedure NEW_LINE    (SPACING : in POSITIVE_COUNT := 1);

   procedure SKIP_LINE   (FILE    : in FILE_TYPE;
                          SPACING : in POSITIVE_COUNT := 1);
   procedure SKIP_LINE   (SPACING : in POSITIVE_COUNT := 1);

   function  END_OF_LINE (FILE : in FILE_TYPE) return
                                               BOOLEAN;
   function  END_OF_LINE                       return
                                               BOOLEAN;

   procedure NEW_PAGE    (FILE : in FILE_TYPE);
   procedure NEW_PAGE                         ;

   procedure SKIP_PAGE   (FILE : in FILE_TYPE);
   procedure SKIP_PAGE                        ;

   function  END_OF_PAGE (FILE : in FILE_TYPE) return
                                               BOOLEAN;
   function  END_OF_PAGE                       return
                                               BOOLEAN;

   function  END_OF_FILE (FILE : in FILE_TYPE) return
                                               BOOLEAN;
   function  END_OF_FILE                       return
                                               BOOLEAN;

   procedure SET_COL     (FILE : in FILE_TYPE;
                          TO   : in POSITIVE_COUNT);
   procedure SET_COL     (TO   : in POSITIVE_COUNT);

   procedure SET_LINE    (FILE : in FILE_TYPE;
                          TO   : in POSITIVE_COUNT);
   procedure SET_LINE    (TO   : in POSITIVE_COUNT);
```

```
   function  COL          (FILE : in FILE_TYPE) return
                                     POSITIVE_COUNT;
   function  COL                                 return
                                     POSITIVE_COUNT;

   function  LINE         (FILE : in FILE_TYPE) return
                                     POSITIVE_COUNT;
   function  LINE                                return
                                     POSITIVE_COUNT;

   function  PAGE         (FILE : in FILE_TYPE) return
                                     POSITIVE_COUNT;
   function  PAGE                                return
                                     POSITIVE_COUNT;

 -- Character Input-Output

 procedure GET  (FILE : in     FILE_TYPE;
                 ITEM :    out CHARACTER);
 procedure GET  (ITEM :    out CHARACTER);
 procedure PUT  (FILE : in FILE_TYPE;
                 ITEM : in CHARACTER);
 procedure PUT  (ITEM : in CHARACTER);

 -- String Input-Output

 procedure GET  (FILE : in     FILE_TYPE;
                 ITEM :    out STRING);
 procedure GET  (ITEM :    out STRING);
 procedure PUT  (FILE : in FILE_TYPE;
                 ITEM : in STRING);
 procedure PUT  (ITEM : in STRING);

 procedure GET_LINE  (FILE : in     FILE_TYPE;
                      ITEM :    out STRING;
                      LAST :    out NATURAL);
 procedure GET_LINE  (ITEM :    out STRING;
                      LAST :    out NATURAL);
 procedure PUT_LINE  (FILE : in     FILE_TYPE;
                      ITEM : in     STRING);
 procedure PUT_LINE  (ITEM : in     STRING);


 -- Generic Package for Input-Output of Integer Types

generic
    type NUM is range <>;
package INTEGER_IO is

   DEFAULT_WIDTH : FIELD       := NUM'WIDTH;
   DEFAULT_BASE  : NUMBER_BASE :=        10;
```

```
    procedure GET  (FILE  : in     FILE_TYPE;
                    ITEM  :    out NUM;
                    WIDTH : in     FIELD := 0);
    procedure GET  (ITEM  :    out NUM;
                    WIDTH : in     FIELD := 0);

    procedure PUT  (FILE  : in FILE_TYPE;
                    ITEM  : in NUM;
                    WIDTH : in FIELD := DEFAULT_WIDTH;
                    BASE  : in NUMBER_BASE := DEFAULT_BASE);
    procedure PUT  (ITEM  : in NUM;
                    WIDTH : in FIELD := DEFAULT_WIDTH;
                    BASE  : in NUMBER_BASE := DEFAULT_BASE);

    procedure GET  (FROM  : in     STRING;
                    ITEM  :    out NUM;
                    LAST  :    out POSITIVE);
    procedure PUT  (TO    :    out STRING;
                    ITEM  : in     NUM;
                    BASE  : in     NUMBER_BASE :=
                                          DEFAULT_BASE);

  end INTEGER_IO;


   -- Generic Packages for Input-Output of Real Types

generic
   type NUM is digits <>;
package FLOAT_IO is

   DEFAULT_FORE : FIELD :=              2;
   DEFAULT_AFT  : FIELD := NUM'digits - 1;
   DEFAULT_EXP  : FIELD :=              3;

   procedure GET  (FILE  : in     FILE_TYPE;
                   ITEM  :    out NUM;
                   WIDTH : in     FIELD := 0);
   procedure GET  (ITEM  :    out NUM;
                   WIDTH : in     FIELD := 0);

   procedure PUT  (FILE : in FILE_TYPE;
                   ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);
   procedure PUT  (ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);
```

```
   procedure GET  (FROM : in     STRING;
                   ITEM :    out NUM;
                   LAST :    out POSITIVE);
   procedure PUT  (TO   :    out STRING;
                   ITEM : in     NUM;
                   AFT  : in     FIELD := DEFAULT_AFT;
                   EXP  : in     FIELD := DEFAULT_EXP);

end FLOAT_IO;

generic
   type NUM is delta <>;
package FIXED_IO is

   DEFAULT_FORE : FIELD := NUM'FORE;
   DEFAULT_AFT  : FIELD := NUM'AFT;
   DEFAULT_EXP  : FIELD := 0;

   procedure GET  (FILE  : in     FILE_TYPE;
                   ITEM  :    out NUM;
                   WIDTH : in     FIELD := 0);
   procedure GET  (ITEM  :    out NUM;
                   WIDTH : in     FIELD := 0);

   procedure PUT  (FILE : in FILE_TYPE;
                   ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);

   procedure PUT  (ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);

   procedure GET  (FROM : in     STRING;
                   ITEM :    out NUM;
                   LAST :    out POSITIVE);
   procedure PUT  (TO   :    out STRING;
                   ITEM : in     NUM;
                   AFT  : in     FIELD := DEFAULT_AFT;
                   EXP  : in     FIELD := DEFAULT_EXP);

end FIXED_IO;
```

```
   -- Generic Package for Input-Output of Enumeration Types

generic
   type ENUM is (<>);
package ENUMERATION_IO is

   DEFAULT_WIDTH   : FIELD    := 0;
   DEFAULT_SETTING : TYPE_SET := UPPER_CASE;

   procedure GET  (FILE : in     FILE_TYPE;
                   ITEM :    out ENUM);
   procedure GET  (ITEM :    out ENUM);

   procedure PUT  (FILE  : in FILE_TYPE;
                   ITEM  : in ENUM;
                   WIDTH : in FIELD       := DEFAULT_WIDTH;
                   SET   : in TYPE_SET    := DEFAULT_SETTING);

   procedure PUT  (ITEM  : in ENUM;
                   WIDTH : in FIELD       := DEFAULT_WIDTH;
                   SET   : in TYPE_SET    := DEFAULT_SETTING);

   procedure GET  (FROM : in     STRING;
                   ITEM :    out ENUM;
                   LAST :    out POSITIVE);
   procedure PUT  (TO   :    out STRING;
                   ITEM : in     ENUM;
                   SET  : in     TYPE_SET := DEFAULT_SETTING);

end ENUMERATION_IO;

   -- Exceptions

   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
   LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

private

  type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end TEXT_IO;
```

F.8.6 Low Level Input-Output

The package LOW_LEVEL_IO is empty.

F.8.a Clarifications of Ada Input-Output Requirements Summary

The Ada Input-Output concepts as presented in Chapter 14 of ARM do not constitute a complete functional specification of the Input-Output packages. Some aspects are not discussed at all, while others are deliberately left open to an implementa- tion. These gaps are filled in below, with reference to sections of the ARM.

F.8.b Assumptions

14.2.1(15): For a sequential or text file, a RESET operation to OUT_FILE mode deletes any elements in the file, i.e. the file is emptied. Likewise, a sequential or text file opened (by OPEN) as an OUT_FILE, will be emptied. For any other RESET operation, the contents of the file is not affected.

14.2.1(7) : For sequential and direct io, files created by SEQUENTIAL_IO for a given type T, may be opened (and processed) by DIRECT_IO for the same type and vice-versa. In the latter case, however, the function END_OF_FILE (14.2.2(8)) may fail to produce TRUE in the case where the file has been written at random, leaving "holes" in the file.

F.8.c Implementation Choices

14.1(1) : An external file is either any VAX/VMS file residing on a file-structured device (disk,tape), a record structured device (terminal, lineprinter), or a virtual software device (mailbox).

14.1(7) : An external file created on a file-structured device will exist after program termination, and may later be accessed from an Ada program.

14.1(13) : See Section F.8.2.1 File Management.

14.2.1(3) : The name parameter, when non-null, must be a valid VAX/VMS file specification referring to a file-structured device; a file with that name will then be created. For a null name parameter, the process' current directory and device must designate a directory on a disk device; a temporary, unnamed file marked for deletion will then be created in that directory.

The form and effect of the form parameter is discussed in Sections F.8.2.1 and F.8.3.1.

Creation of a file with mode IN_FILE will raise USE_ERROR.

14.2.1(13): Deletion of a file is only supported for files on a disk device, and requires deletion access right to the file.

14.2.2(4): No check for DATA_ERROR is performed in case the element type is of an unconstrained type.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below:

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	(1..125 => 'A', 126 => '1')
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	(1..125 => 'A', 126 => '2')
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	(1..63 => 'A', 64 => '3', 65..126 => 'A')

TEST PARAMETERS

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	(1..63 => 'A', 64 => '4', 65..126 => 'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..123 => '0', 124..126 => "298")
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..120 => '0', 121..126 => "69.0E1")
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1 => '"', 2..63 => 'A', 65 => '"')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1 => "'", 2..63 => 'A', 64 => '1', 65 =>"'"))
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..106 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2_097_152
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	VAX11
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	35
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	200000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	15
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	ILLEGAL!@#$%^
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	ILLEGAL&()_+=
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32768

TEST PARAMETERS

Name and Meaning	Value
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	32767
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	32_768
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-200000.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	126
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 => "2#", 3..123 => '0', 124..126 => "11#")

Name and Meaning	Value
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3 => "16#", 4..122 => '0', 123..126 => "F.E#")
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 => '"', 2..125 => 'A', 126 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2_147_483_648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	VAX11
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFF#
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	2_097_152

TEST PARAMETERS

Name and Meaning	Value
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	VAX11
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	0.000_001

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 25 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

A39005G: This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

B97102E: This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

CD2A62D: This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

CD2A63B, CD2A63D, CD2A66B, CD2A66D, CD2A73A..D, and CD2A76A..D (12 tests): These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)).

CD2A81G and CD2A83G: These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this need not happen, and the main program may loop indefinitely (lines 74 and 85, respectively).

CD2B15C and CD7205C: These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

CD7105A: This test requires that CALENDAR.CLOCK return values that differ by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the average of such differences that must be at least SYSTEM.TICK --particular differences may be less (line 29).

WITHDRAWN TESTS

CD7205D: This test checks an invalid test objective: it expects the 'STORAGE_SIZE length clause for tasks to reserve the specified amount of storage as though for a collection, with the task type's objects using part of the storage for their activation, instead of expecting the specified storage to be what is reserved for the activation of any task object.

CE2107I: This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid (line 90).

CE3111C: This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

CE3301A: This test contains several calls to END_OF_LINE and END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, and 136).

CE3411B: This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.